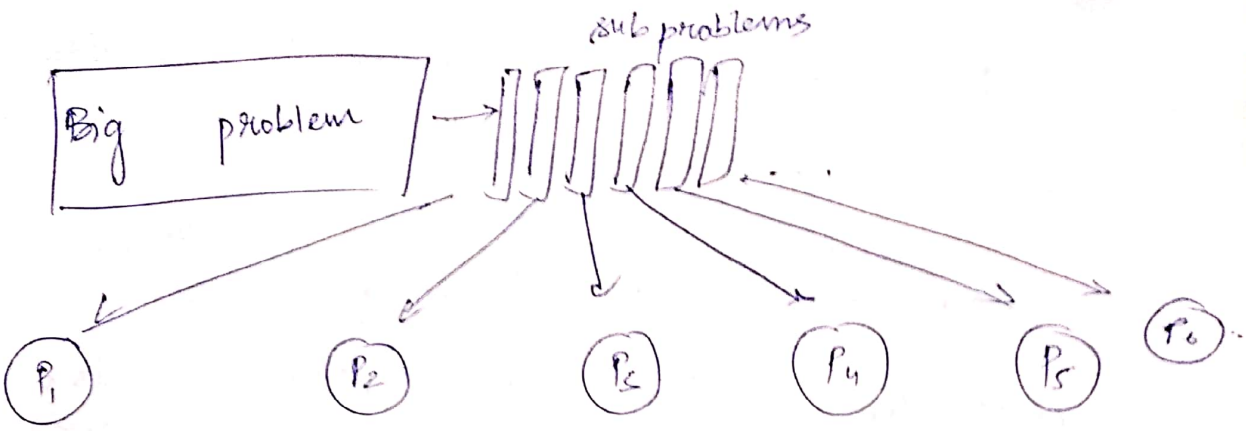
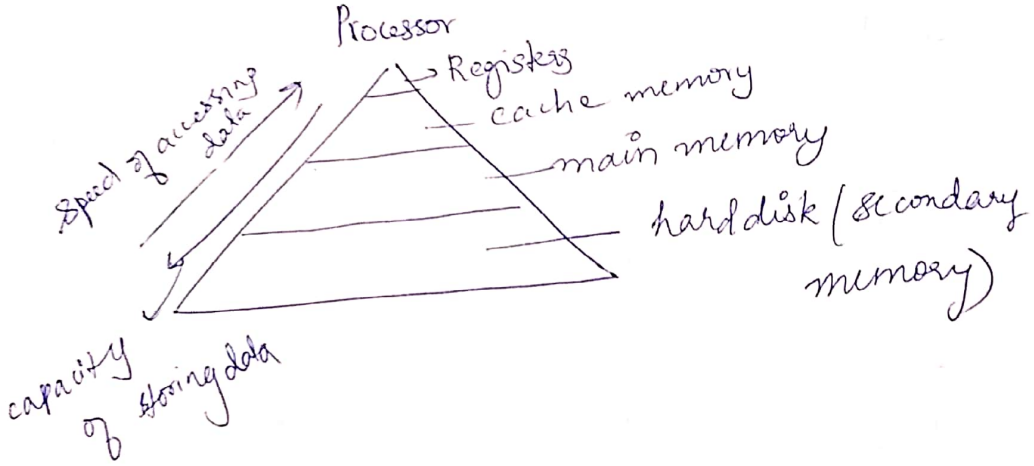
Parallel computing



memory hierarchy

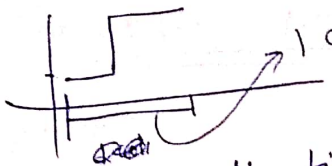


$$\text{Performance}_x = \frac{1}{\text{Execution Time}_x}$$

x is n times faster than y

$$\therefore \frac{\text{Performance}_x}{\text{Performance}_y} = \frac{\text{Execution}_y}{\text{Execution}_x} = n$$

Clock cycles



→ It's better to express execution time in ~~seconds~~ cycles rather than in seconds

→ clock ticks → start and end of a cycle.

→ cycle time = time between ticks = seconds per cycle

→ ~~cycle rate~~ clock rate = cycles per second.

→ 1 Hz = 1 cycle/sec, 1 MHz = 10^6 cycles/sec

Clock cycle time = 1 / clock rate

Performance Eq ①

$$\frac{\text{Seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

CPU execution time for a program = CPU clock cycles for a program × clock cycle time

or

= CPU clock cycles for a program / clock rate

Performance :- The capacity to reduce to execution time of a program with some additional resources.

computer performance is based upon.

(i) Response Time :- the time it takes ~~from~~ for the program to start and finish, it includes accessing disk and memory, waiting for I/O and other processes, OS overhead.

(ii) Throughput :- It talks about how much amount of the data is being processed in a given time

or

It talks about how ~~much~~ many processes or jobs can the machine run at once.

Q) Execution time initial = 10 s

Clock rate = 4×10^9 cycles/sec

Execution time new = 6 s

1 cycle/second

10 s = $\frac{\text{CPU clock cycles A}}{4 \times 10^9 \text{ cycles/sec}}$

CPU = 1 ns

CPU clock cycles A = $4 \times 10^9 \times 10$ cycles

CPU clock cycles B = $1.2 \times$ CPU clock cycles A

$\therefore$ ~~Execution time~~ $6 = \frac{\text{CPU clock cycles B}}{\text{clock rate}}$

Clock rate = $\frac{1.2 \times 4 \times 10^9 \times 10}{6}$

$= 8 \times 10^9$

$= 8$ GHz

Performance Eg (2)

CPU execution time = (Instruction count for a program × average CPI (CPU clock cycles) × clock cycle time)

units = seconds/program

$$\therefore \text{CPU Ex} = \text{IC} \times \text{CPI} \times \text{clock cycles}$$

CPI → clock cycles per Instruction

$$\text{CPU exec time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Q)

$$\frac{\text{CPU execution A}}{\text{CPU execution B}} = \frac{I \times 2.5 \times 2}{2I \times 2.5 \times 1.2}$$

$$\Rightarrow \frac{\text{CPU execution B}}{\text{CPU execution A}} = 1.2 = \frac{\text{performance of A}}{\text{performance B}}$$

CPU clock cycles = Instruction count for a program × average CPI.

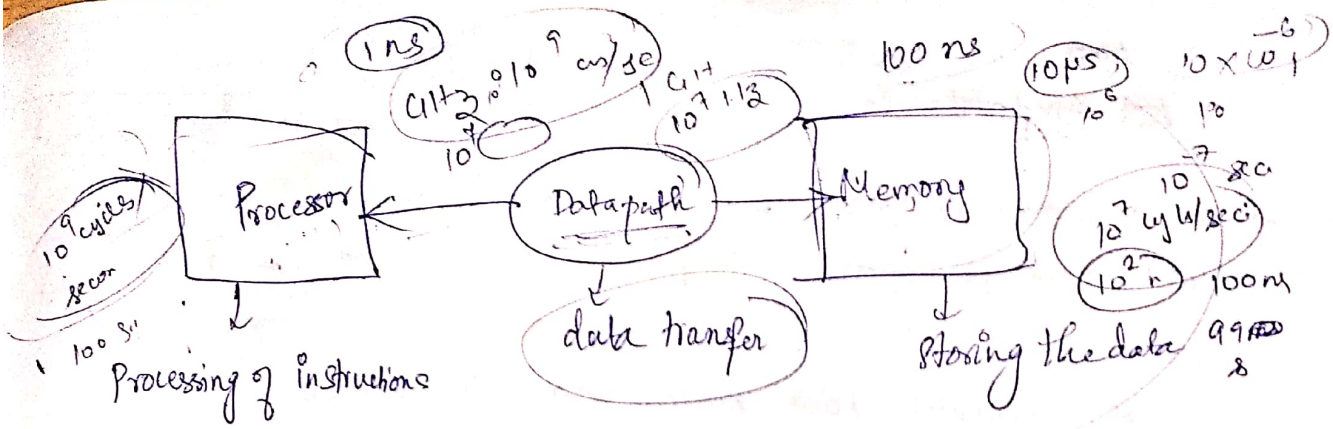
$$10 = (5) \times c$$

$$= CPI_1 = 2$$

$$9 = (6) \times CPI_2$$

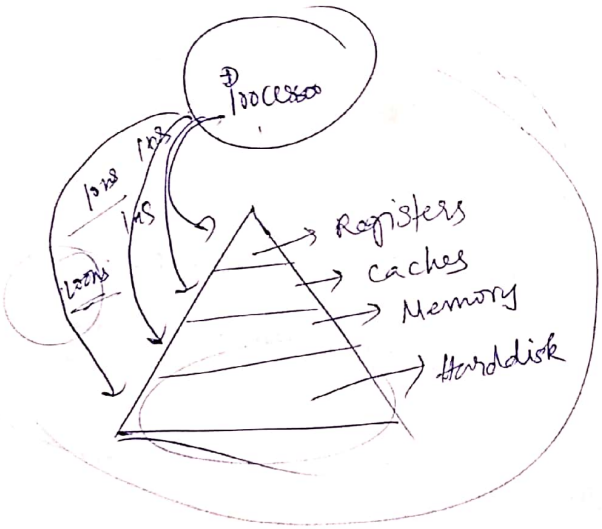
$$CPI_2 = 1.5$$

CPU Execution time :- It refers to the time in which CPU processes the task or executes the given task (excludes I/O, & running other programs) this is simply referred to as **CPU time**



→ Memory is very slow compared to processor

→ Bottlenecks occur as the rate at which memory sends data is less than the rate at which the processor processess data.



Multiplicity → more no of processors, more, data paths, more memory.

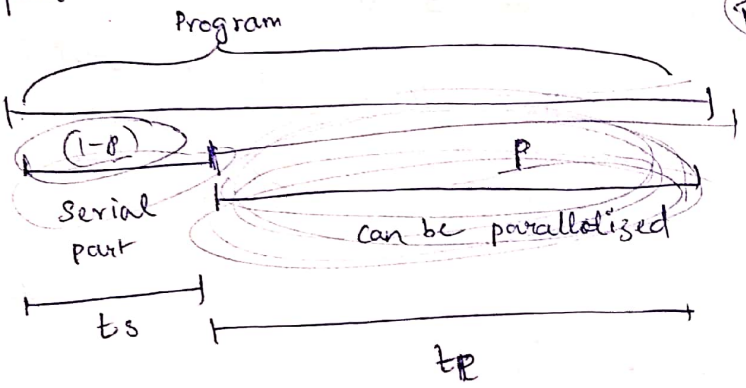
Implicitly → compilers will take care of parallelism

Indirect → parallelism is taken by the programmer.

Amdahl's law

→ In a program there will be serial processes and some part of the serial process which can be parallelized.

→ So the Amdahl's law states that there will be a performance improvement when we parallelize the part of program, and the time that can decreased for processing depends on this part



$$T_{new} = (1-P)T + \frac{P(T)}{n}$$

$$= (1-P)T + P\frac{T}{n}$$

Speedup = $\frac{\text{Performance for the task using the enhancement when possible}}{\text{performance for the task without using enhancement}}$

$$= \frac{T}{(1-P)T + P\frac{T}{n}}$$

(or)

$$= \frac{\text{Execution time taken for a task without using enhancement}}{\text{Execution time taken for a task with using enhancement when possible}}$$

let the P be the fraction of the task to be enhanced.

$T_{old} = T$

$T_{new} = T_{old}(1-P) + \frac{T_{old} P}{n}$

$(T_{new} = T(1-P) + TP)$

$$\frac{T_{old}}{T_{new}} = \frac{1}{(1-p) + \frac{p}{n}}$$

→ here n is the no. of processors

→ p is fraction of the task that can be parallelized

$$\text{SpeedUp} = \frac{1}{(1-p) + \frac{p}{n}} \quad \text{as } n \to \infty$$

$$5 = \frac{T_{old}}{T_{new}} \quad T_{new} = 20s$$

Ex = $T_{old} = 100$

$$20 = 20 + \frac{80}{n}$$

$$\frac{80}{n} = 0$$

Ex → Speedup = 5, $T_{old} = 10$, $n = 5$

$T_{old} = 10s$

$$T_{new} = 5 + \frac{5}{5}$$

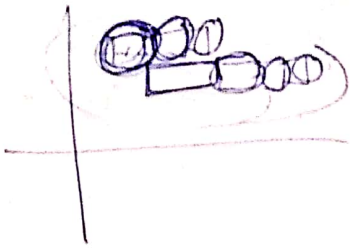
$$= 6$$

$$\text{Speedup} = \frac{1}{5 + \frac{5}{5}} = \frac{1}{5+1} = \frac{1}{6}$$

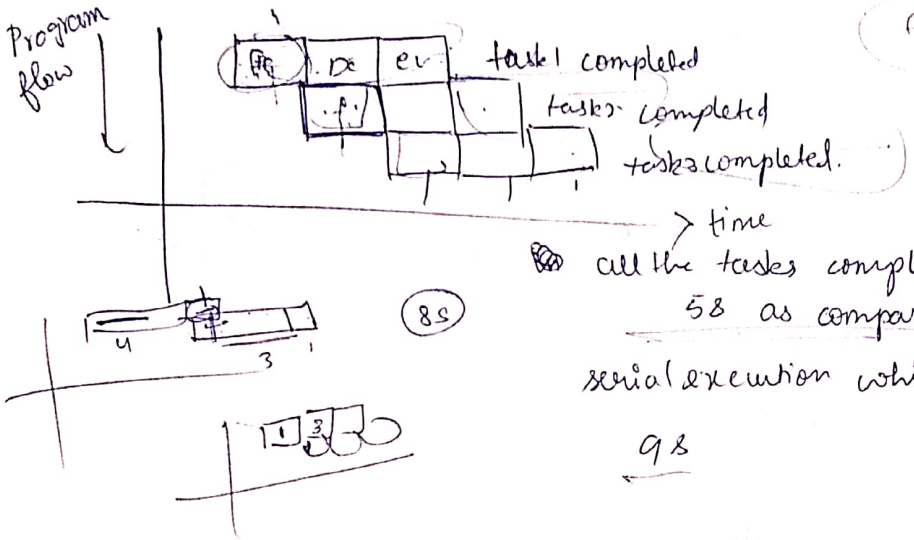
$$\therefore \text{Speedup} = \frac{10}{6} = 1.6$$

Q) How process relies on different process to achieve parallelism and acheive better performance.

→ Pipelining
→ Superscaling
→ Very long instruction word



Pipelining :- It overlaps various stages of instruction during execution to acheive performance.

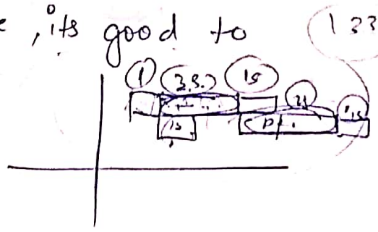


All the tasks completed in 5s as compared to serial execution which takes 9s

→ processors have relied on pipelining technique to improve execution rate.

→ to increase speed of a single pipeline, one would break down the tasks into smaller units.

→ to further improve the execution rate, its good to use multiple pipelines.



Problems in Pipelines

→ The pipeline get slowed down due to slowest step or stage

• to solve this, a conventional processor relies on a very deep pipeline (upto 20 stages)

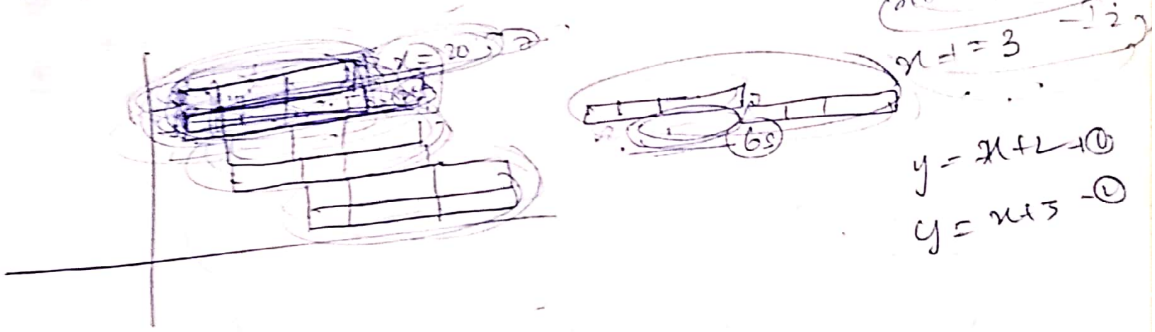
→ even then as every instruction not only consists of add or mult, it may even have jump conditions based on the if and else constructs, this jump requires a accurate branch prediction.

→ If there is a misprediction, then the penalty of such prediction grows through the depth of the pipeline and a large number of instructions have to be flushed.

Superscalar

→ one simple way of dealing with problems in pipelining is to use multiple pipelines

→ same time 2-2 instructions will be fetched (i.e) in pipeling method we have single stage instructions being fetched, but here 2 instructions are fetched at a time.



issues in Superscalar

→ not all functions are kept busy at all times.

→ ~~[illegible]~~ If during a cycle, no functional units are utilized then it refers to the vertical waste

→ If during a cycle, only some of the functional units are utilized, this refers to as a horizontal waste.

→ data dependencies may waste some functional units.

Scheduling of instructions is determined by a number of factors

→ true data dependency - the result of one op is an i/p to either one

→ Resource dependency : one resource required by 2 o/p

→ Branch dependency - For conditional branch instruction destination, is only known at the point of execution

→ In simpler model, instructions can be issued only in the order in which they are encountered, (i.e) if the second instruction has a data dependency with the first, then the second instruction is not issued along with the first instruction. this is called in-order issue.

→ In a more aggressive model, if the 2nd instruction has a data dependency with 1st and there is a 3rd instruction which doesn't have any dependency with 1st and 2nd instruction. then in the first clock cycle the 1st and 3rd instruction can be clubbed, this is called a dynamic issue.

→ There is a hardware tool called scheduler which looks into the long queue and selects appropriate instructions to execute concurrently based on the dependencies

To overcome the dependency issues present in the superscalar we use VLIW

VLIW (very long instruction word)

→ these processors came into existence inorder to address the complexity of superscalar processor and its scheduler

→ VLIW processors rely on compile time analysis

→ analyse what instructions to bundle together that can be executed concurrently.

→ these instructions are packed and dispatched together as a single instruction longword hence VLIW.

VLIW

Issues with ~~RISC~~ VLIW

→ compiler has a bigger context from which it has to analyse

→ ~~S~~ compilers however do not have runtime info

→ Branch and memory prediction is more difficult

→ VLIW performance is highly dependent on the processor

→ A typical VLIW implements 4-way or 8-way parallelism

Limitations of Memory system Performance

→ memory speed is less as compared to processor speed ∴, hence bottlenecks occur

→ memory system performance is largely captured by two parameters, latency and bandwidth

→ latency is the time taken ~~by~~ ~~or the memory~~ from the issue of a memory request to the time the data is available at processor

for example, to fetch a ~~from~~ b value of A, the processor sent a request at the 0th seconds and the data b is available at processor at 2nd second ∴ the latency here is 2s

bandwidth ⇒ It's the rate at which the data is being pumped to the processor

Ex:- processor speed = 1 GHz (1 ns clock)

Dram latency = 100 ns (no caches)

peak performance (GFlops) = cpu speed in (GHz) × (CPU instruction per cycle)

$$GFlops = \frac{cycles}{second} \times \frac{instructions}{cycle} = \frac{instructions}{second}$$

↳ Processor executes 4 instructions per cycle.

∴ peak performance is = 4 GFlops = 4 × 1 GHz

= 4 GFlops

But due to the memory latency, the processor has to wait for $\frac{100 \text{ clock cycles}}{1}$ to process the data

∴ ~~bandwidth~~ or

It depends on how much fast the memory pumps in the data to the processor, hence

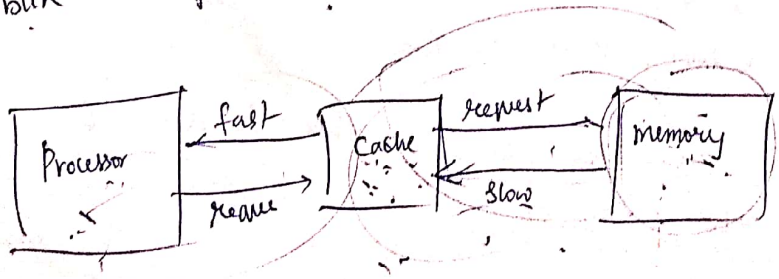
$$\text{bandwidth} = \frac{1}{100 \text{ ns}} = \frac{1}{100 \times 10^{-9}} = 10^7 \text{ Flops}$$

$= 10 \times 10^6$ Flops

= 10 MFlops

Improving effective memory latency using caches

↳ caches are small and fast memory elements b/w the processor and the D-Ram



$\frac{1}{10^9}$ = 1 ns

→ If a piece of data is repeatedly used, the effective latency of the memory system can be reduce by cache

→ The fraction of data references satisfied by the cache is called cache hit ratio

→ Initially some things are not present on the cache, in the intial request the cache will not have the data and hence its a miss, therefore cache will get the data from main memory and store it, in the next request the cache will be able to provide the data to processor and hence its a hit.

∴ hit ratio = $\frac{\text{no of hits}}{\text{no of hits + no of misses}}$

consider the before page example and there is a cache of 32 kB

→ So first we need to fetch the two matrices from the main memory. So for that there will be some time it takes

lets calculate it (delay) → $2 \times 32 \times 32 \times 100\text{ns delay}$

2 matrices, matrix

$\Rightarrow 2 \times 32 \times 32 \times 100 \times 10^{-9}$

$\cong 200\mu s$

→ now multiplying 2 matrices will take $2n^3$ operations.

∴ the total operation = $2 \times 32^3 \cong 64k$ operations

∴ the processor performs, 4 instruction/operations per cycle

∴ total no of cycle required by the processor = $\frac{64k}{4}$

(for 16 μs) = 16k cycles

now the peak computation rate = cpu speed × no. of instructions or operations per cycle

total delay = 200 μs + 16 μs
= 216 μs

∴ cpu speed = 1 / 216 μs

∴ peak computation rate = 64 instruction/cycle × 1/216 μs

= 64 × 1000 / 216 × 10^-6

= 10^9 / 3.375

= 0.2962 × 10^9

= 296 × 10^6 F

= 296 MFlops

Impact of memory bandwidth

→ memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.

→ memory can be improved by increasing the size of memory blocks.

→ spatial locality → storage of consecutive words in the cache

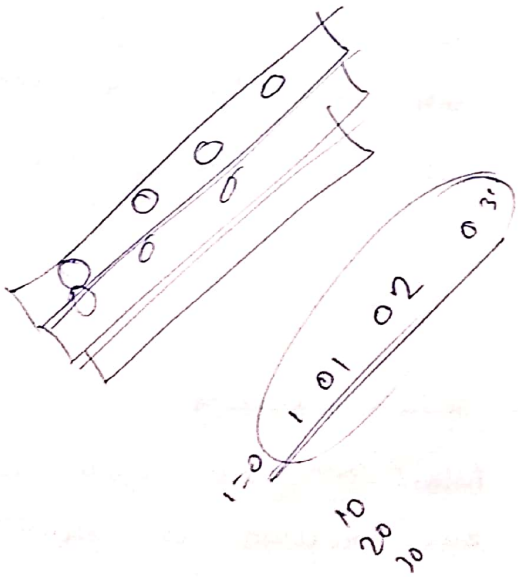
→ temporal locality → storage of non consecutive words in the cache.

Some other ways of hiding memory latency

→ prefetching → It's a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory even before

→ multi-threading → creating different threads in order to execute different operations which are not dependent on each other

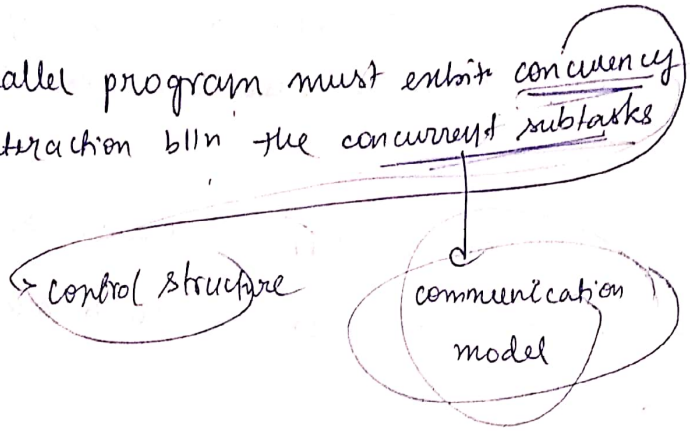
→ Spatial locality → It's a technique in which we bring in all the data into the cache from the main memory.



Explicitly parallel programs

→ this type of parallelism is controlled by the programmers.

→ a parallel program must exhibit concurrency and interaction bw the concurrent subtasks



→ parallelism can be expressed at various levels of granularity.

from instruction level to process level

control structure of parallel programs

→ processing units parallel computers either operate under a → centralised control (or) → work independently

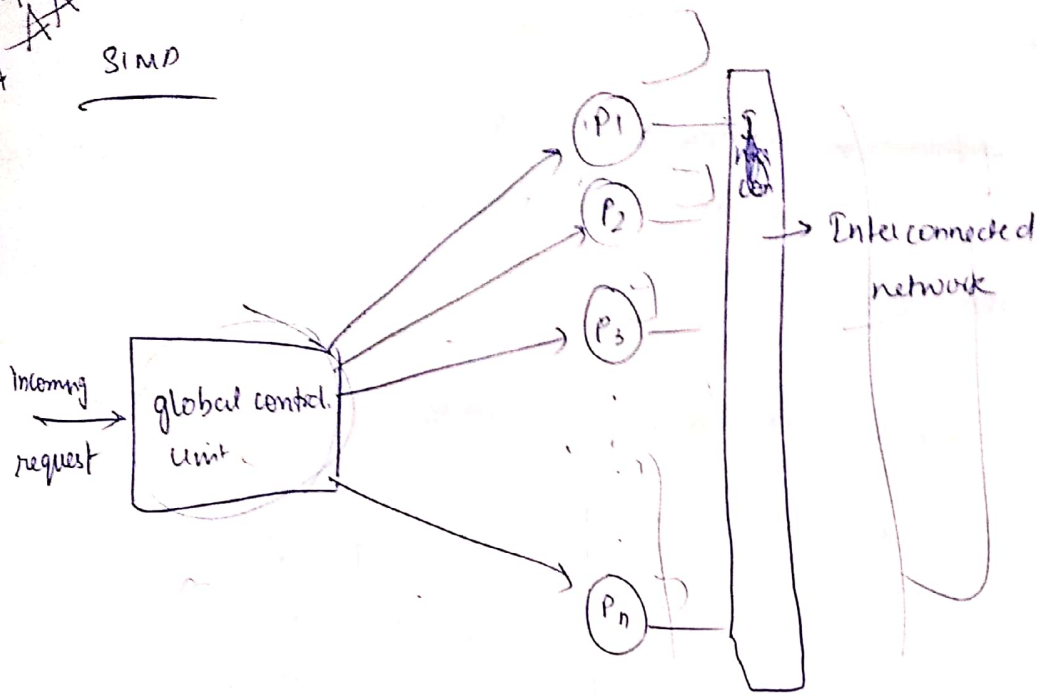
→ If there is a single control unit, that sends instructions to various processors that work on different data, this model is refered to as SIMD → Single instruction stream, multiple data stream.

→ If ~~the~~ ~~data~~ p

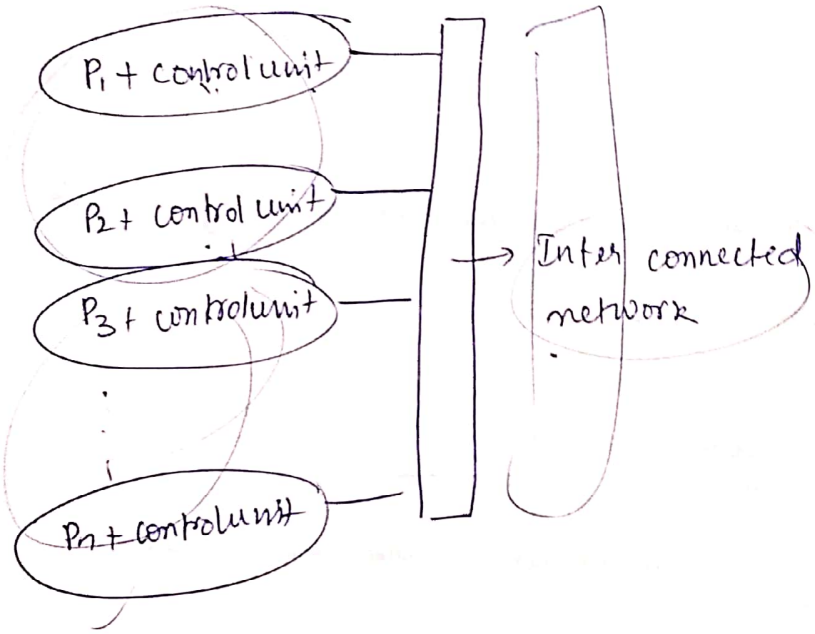
MIMD → each processor has its own control unit, each processor can executed different instructions on different data items

multiple instruction stream, multiple data stream.

SIMD



MIMD



SIMD

→ example Illac IV, MPP, DAP

→ sometimes we ~~don't~~ don't want some operations on certain data items, an activity mask determines whether a processor should participate the computation or not.

→ single program, on different processors

MIMD

→ multiple programs on multiple processors.

→ Single program multiple data streams.

→ Sun ultra servers, IBM servers

SIMD vs MIMD

→ SIMD requires less hardware than MIMD because executing on the same instruction and also there is one global control unit whereas on MIMD each processor has its own control unit hence the hardware complexity increases.

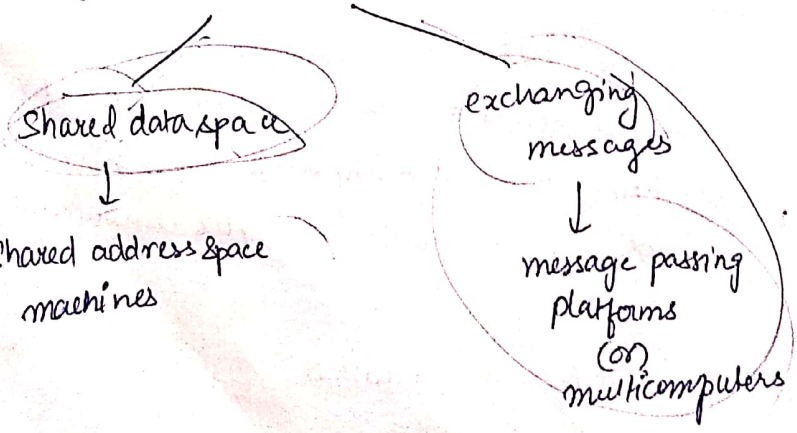
→ SMID are specially designed hence they tend to be expensive as compared to MIMD

→ Not all the applications are naturally suitable with SIMD

Communication Model

How does the communication b/n the processors.

→ two forms of data exchange

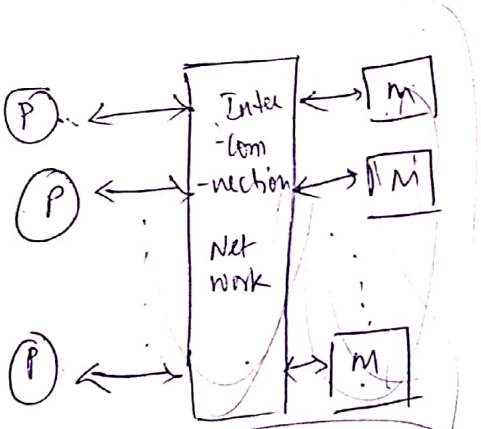


Shared address space platforms

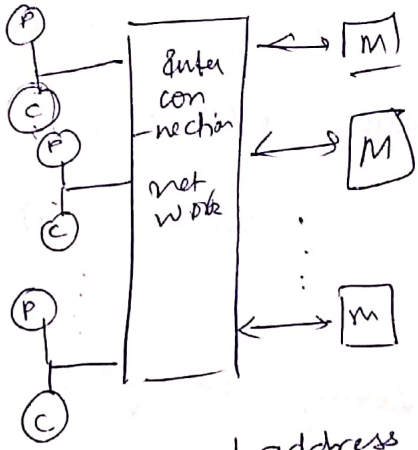
→ partial or all the data is available to all processors

→ processors interact by modifying the data in in shared address space

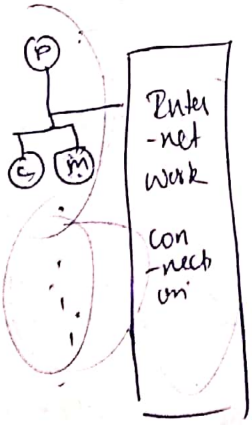
→ If the time taken by the processor to access any memory in the global space or local is identical then it's reffered uniform memory access (UMA) else it's reffered as NON uniform memory access (NUMA)



UMA shared address space



UMA shared address space with cache



& NUMA shared-address space with local memory

NUMA and UMA

→ The distinction bwn NUMA and UMA is very much important as in NUMA each and every processor has its own cache and local memory whereas on the other side in UMA, the processors have shared memories, hence an algorithm should be defined in such a way that the processor can get the data efficiently

→ programmig of these platforms is easier since read and write are visible to other processor.

→ there should be a co-ordination of read-write data to the shared-address-space as there might be conflicts when one processor is reading from the memory and at the same time another process might be writing into it.

→ cache coherence problems

Message passing platforms

→ We are passing messages from one processor to another as one processors output may serve as an input other one.

→ These platforms are especially used in cluster work stations

→ Ex MPI, PVM ← these libraries

→ programmig of these platforms are done through

MPI V Shared-Address Space

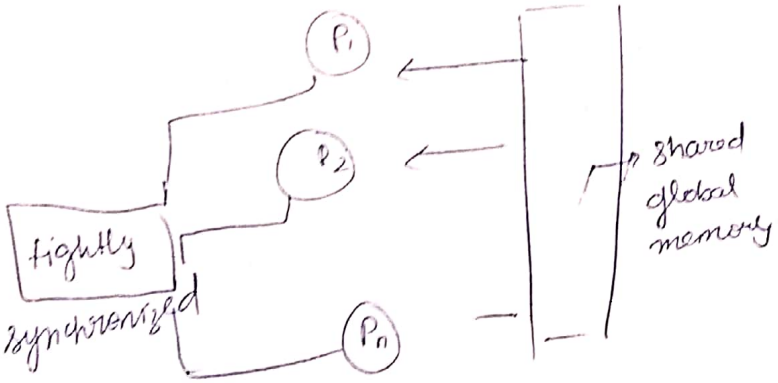
→ MP requires little hardware support, other than network whereas Shared-address-space requires a lot of hardware support (i.e) there should be controllers which make sure that there are no conflicts while reading and writing data to it.

Physical Organization of Parallel platforms

PRAM -> Parallel random access machine

→ pram consists of p processors and they all have access to the global shared memory.

→ all the processors have a common clock but can execute different instructions in each cycle



Depending on how simultaneously it access the memory PRAMs are classified as

→ Exclusive read - Exclusive write EREW PRAM
→ Concurrent read - Exclusive " CREW PRAM
→ Exclusive read - Concurrent write ERCW PRAM
→ Concurrent " - Concurrent " CRCW PRAM

Concurrent

↓

Common values are allowed to write

↓

Arbitary process will be given ability to write

↓

Priority is given to processes to write

↓

Sum of all the data is written in to the memory

→ proccessors and memories are connected via switches.

→ these switches must operate in O(1)

→ interconnection networks carry data b/w processors and to memory

→ interconnects are made of links and switches

→ static interconnects are interconnect which connect point to point these are alge direct connections

→ dynamic network are built using switches and links. these are Indirect networks

Network Topologies

→ Buses

- distance bin any two nodes is O(1)
→ all the entities involved share the same bus

Crossbars

→ Is a pxm network

Thread

→ A thread is a single sequential flow of control within a program.

→ multiple threads can be executed in parallel on computer systems.

→ This occurs by time slicing, time-slicing is based on non-priority scheduling.

→ threads typically share memory and other resources directly

→ The communication overhead is less they use the shared memory space.

→ user level → user level will take care.

→ kernel level → operating system will take care

→ the threads can be use via pthread header file and can be compiled using gcc itself.

→ Stack will be created for each thread

→ threads will share the shared-address-space and have their own memory, hence it is a leads to poor performance

→ Threads are portable because every processor are capable of handling the functions of threads software.

→ inherent support for latency hiding → one thread is independent of other, for example while one thread is doing i/o operations, the other thread can do some other operations

→ scheduling and load balancing

→ ease of programming

~~Thread creation and termination~~

Thread safeness

→ It's the ability to execute multiple threads simultaneously without clobbering shared data or creating race conditions.

Why pthreads?

→ performance gains

→ cost of managing a process is more than managing a thread as it requires fewer system resources.

→ on modern cpu's, pthreads are suitable for parallel programming

pthread API

1) thread management :- functions that directly work on threads

→ creating
→ detaching
→ joining

2) mutex :- functions that deal with synchronization called a mutex, this includes creating, destroying, locking, unlocking mutexs

5- conditional variables → functions that address communications b/w threads that share a mutex.

→ This class includes functions to create destroy wait and signal based on ~~mutexes~~ specified variables

thread creation

pthread_create (thread, attr, start_routine, arg)

- thread: a unique identifier
- attr: set thread attributes
- start_routine: function will be executed once the thread is created
- arg: arguments to start_routine

thread termination

→ the thread is terminated when it's returned to main from the start routine

→ via pthread_exit() → this is explicit

→ one thread is cancelled by other thread
pthread_cancel()

→ the entire main process is exited due to exit routine.

pthread_exit is use explicitly exit a thread; it's generally called after the thread has completed its work and no longer required to exist

→ pthread_join() tells to execute the thread.

Synchronization primitives in threads

→ If there is no sync b/n threads, it may result in to unwanted solutions, hence we need synchronization b/n threads.

mutual exclusion

```
if (mycost < best cost)
{
    best cost = mycost
}
```
} critical sections

→ critical segments in pthreads are implemented using mutex locks

locked / unlocked

→ at any point of time 1 thread can have a mutex lock.

→ ~~A thread~~.

pthread_mutex_lock

pthread_mutex_unlock.

Producer Consumer using locks

→ manage shared memory access

→ checking if buffer is full (producers)

→ checking if buffer is empty (consumers)

```
#NUM_treads = 2
int buffer [10];
int count = 0;
int main() {
  Pthread_t th[Num_threads]

  for (int i=0; i< num_threads; i++)
  {
    if (i%2==0)
    pthread_create(&thread[i], NULL, &producer, NULL)
    else
    pthread_create(&th[i], NULL, &consumer, NULL)
  }
```

```
void* producer (void* args)
{
    int x = rand() % 100;
    buffer[count] = x;
    count++;
}
```

Types of mutex

→ Normal mutex :- are this mutex deadlocks if a thread that already has a lock tries a second lock on it

→ recursive → single thread is allowed to lock a mutex as many time it wants.

Error check mutex → reports an error when a thread with a lock tries to lock it again.

Proc : 1 GHz

$\frac{1}{10^9}$ s

1 cycle = $\frac{1}{10^9}$ s

1 cy ⟶ $\frac{1}{10^9}$ s

1000 ⟶ x

$x = \frac{100}{10^9}$

$x = \frac{1}{10^7}$ s

C = $\frac{1}{10^9}$ s

2 × 4 = 8 words

8 × 10^9 s

Cycle / sec

inv / sec

1 cy $\frac{1}{10^7}$ s

$\frac{1}{10^7}$ s

10^7 sec

10^7

CP = Cycle / second